一种新型高效的算法级容错技术及实现

王睿 姚二林 陈明宇 谭光明

摘要 随着高性能计算系统规模的不断扩大，节点失效愈加频发。传统的容错技术大都基于检查点（checkpoint）方式。但是，检查点技术的开销随着系统规模的扩大而不断增加，在百亿亿次（Exaflops）规模下其容错效率难以满足系统需求。算法失效恢复技术相比检查点方式具有更高的效率。然而，该技术依然基于停等模式。对于大规模系统，停等模式在很大程度上会影响程序的并行效率。本文提出了一种非停等的算法级容错策略——热替换策略。在程序运行过程中若发生节点失效，不用停等恢复失效节点上的数据，而用冗余节点替换失效节点，使计算能继续进行。最终的正确结果可以通过一个线性变换求出。为了论证方案的有效性，我们结合 MPICH 的容错特性实现了容错的 High Performance Linpack (HPL)，并评估了方案的性能。实验结果表明，即使在小规模下，我们的方案的性能也明显优于算法失效恢复技术。

关键词 高性能计算；检查点；算法容错；Exaflops

1 引言

1.1 容错技术研究的意义

随着高性能计算机系统计算能力的不断增加，系统的规模也随时间呈指数上升趋势。最新的超级计算机排行（Top500）统计数据显示，目前世界上最快的超级计算机含有的处理器核数已达到 10 万级别[2]。按照当前的技术趋势发展，下一代 E 级计算机的规模将突破 100 万个核[1]。在构建下一代超级计算机的同时，系统的可靠性问题将越来越突出。

一方面，随着系统规模的增大，系统的平均中断间隔时间（mean-time-to-interrupt, MTTI）越来越短。卡内基梅隆大学的 吉普森（Garth Gibson）等人基于对洛斯阿拉莫斯国家实验室（Los Alamos National Laboratory）十年内超级计算机失效数据的分析，发现超级计算机系统的出错频率正比于其中包含的处理器个数[11, 12]，如图 1 所示。

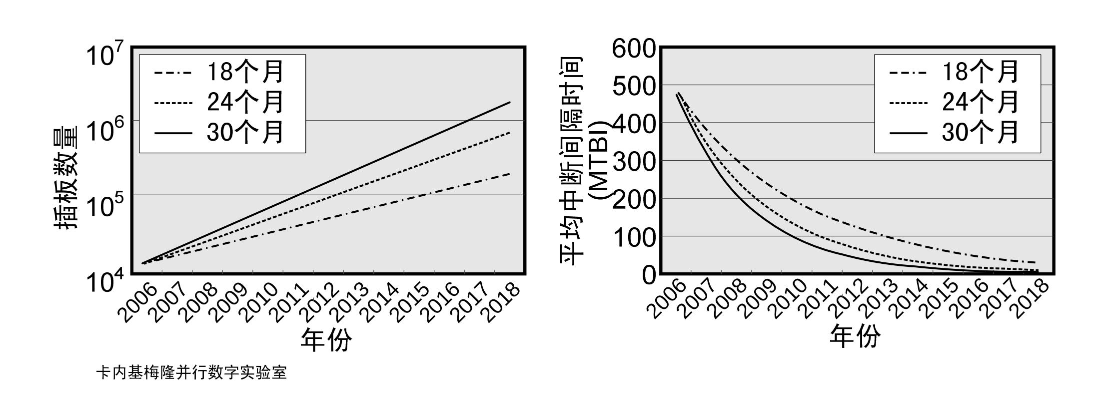


图 1. 系统的平均中断间隔时间和处理器个数之间的关系[11, 12]

图中三条曲线分别代表每个插板上处理器核数量按照 18、24、30 个月翻一番的模型增长时相应的不同统计预测结果（图 2 亦是如此，不另行注明）

另一方面，高性能应用在数据规模、计算复杂度和运行时间上依然不断增长。对于很多大规模科学计算来说，高性能计算系统的平均中断间隔时间已经小于程序的执行时间。因此，

必须采用高效的容错方式，提高系统的可靠性，以满足应用的需要。

1.2 容错技术研究现状

传统的容错技术大都基于检查点，即周期性地保存系统的状态，建立检查点并写入磁盘，系统如果出错即回滚到上个检查点，恢复后重新开始计算。但是，随着高性能计算机系统的计算能力和磁盘的读写能力之间差距的不断增大，检查点容错方式在百亿亿次（Exaflops）规模下容错效率难以满足要求。基于计算机故障数据集中（Computer Failure Data Repository，CFDR）[13]统计数据的分析，吉普森等人预测按照现有的技术趋势发展下去，检查点方式在未来超级计算机系统中的容错效率将会趋于零[8, 11]，如图 2 所示。

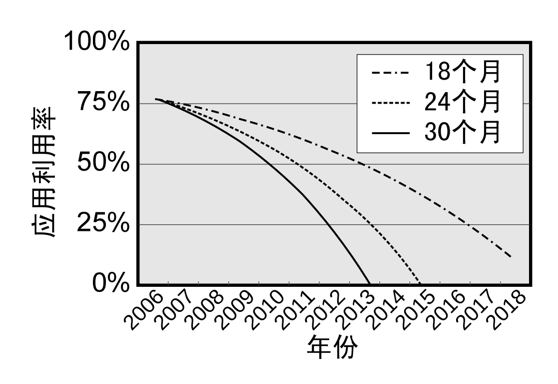


图 2. 吉普森等人对检查点效率的预测[11]

系统级的检查点方式对用户透明，但当系统规模很大时，带来的开销是惊人的。检查点容错方式的主要开销在于：(1). 节点失效需回滚，回滚期间所作的运算作废；(2).需要周期性地向磁盘写入检查点，而外存的读写带宽远低于内存的读写带宽；(3).由于采用停等（stop-and-wait）的容错模式，一个节点失效，全系统都要停下来等其修复。

普朗克（J. S. Plank）等人提出的无盘检查点(diskless checkpointing)技术[14]，使用高速的内存代替低速的磁盘来存储检查点，并对各个计算节点的数据进行编码，存入额外的冗余节点。无盘检查点技术的可扩展性优于检查点方式，但仍然需要回滚和停等的开销。算法失效恢复技术（ABFT recovery）[3-5, 15-18]则更进一步，可以做到不回滚。尽管该技术对应用程序不透明且不具有通用性，但它仍可覆盖较广的应用面，如 ScaLapack[4, 15]、HPL[16]、PCG 求解[5, 17]以及 PETSc 中的线性方程组迭代求解法[18]等。算法失效恢复技术最大的优点是不需要额外的时间保存检查点。冗余数据在计算过程中被同步更新，因此出错也不需要回滚。此外，由于冗余数据在内存中，该方案不需要读写低速外存。实验表明，该技术的性能要优于传统的检查点技术[5, 17]和无盘检查点技术[18]。

然而，算法失效恢复技术仍然基于停等的容错模式，即在程序执行过程中，即使只有一个进程失效，所有运行中的进程都要停下来等待失效进程数据的恢复。根据阿姆达尔（Amdahl[1]）定律，程序的加速比受限于程序中的串行部分。对于大规模系统，停等模式在很大程度上会影响程序的并行效率。

研究一种非停等的、应用层能感知失效的容错策略成为未来容错技术发展的一个新方向。它所面临的挑战主要有：(1). 应用层感知失效需要对应用的算法有较深的理解，对容错技术的实施造成了一定的困难；(2). 如何设计编码模式以保证编码的高效和数值稳定性也是一个待解决的问题；(3). 非停等的容错模式对容错系统的支持提出了更高的要求，现有 MPI[2] 实现的容错特性大都基于检查点方式，实现应用层容错需要 MPI 新的支持。

1.3 本文的主要工作和贡献

基于对算法失效恢复技术的研究，本文提出了一种新型高效的算法级容错方案——算法失效热替换（ABFT hot-replacement）方案。当程序执行过程中发生节点失效，该方案不用

[1] IBM 的杰出架构设计师
[2] Message Passing Interface，一种基于消息传递的并行程序设计标准

停等恢复丢失的数据，而用冗余数据替换丢失的数据，使计算能立刻继续进行下去。在程序执行的最后，正确结果可以由替换后得到的中间结果经过一个简单的线性变换求出。

为了验证方案的正确性，我们基于该方案并结合 MPICH2 新的容错特性，实现了容错的 HPL[3]，并评估了该方案的性能。实验结果显示，即使在小规模下，我们的方案相对算法失效恢复技术方案也有明显的性能优势。

1.4 本文的组织结构

本文共分六章：第一章介绍了本文的研究意义和容错技术的研究现状；第二章综述了相关工作，阐述了算法失效恢复技术，并简要介绍了 MPICH2 对算法级容错的支持；第三章提出了非停等的算法失效热替换方案并讨论了处理多次失效的情况；第四章详细说明了算法失效热替换方案的实现细节；第五章评估了算法失效热替换方案的性能并验证了方案的正确性；第六章对下一步工作进行展望。

2 相关工作

本章先介绍算法容错的基本思想，在此基础上阐述了算法失效恢复技术的容错策略，最后讨论了实现算法级容错技术需要 MPI 提供哪些支持。

2.1 算法失效恢复技术

算法容错（Algorithm Based Fault Tolerance，ABFT）[7]的概念由黄光华（音译，K.H. Huang ）和阿布拉罕（J. A. Abraham）于 1984 年第一次提出。当时是应用在大规模集成电路瞬态错误的检测、定位并修正等。算法容错的核心思想是先将原始数据进行编码变换，然后重新设计算法流程使得冗余数据在计算过程中被同步更新，从而可以在计算中检测和纠正错误。该方案并不是通用的，因为并不是所有的应用都能使得冗余数据和计算数据被同步更新。算法容错技术主要是针对一类包含特定矩阵运算的应用，如矩阵加法、乘法、内积、LU 分解以及转置运算等。

算法失效恢复技术[3-5, 15-18]是陈子忠（Z. Chen）和多加拉（J. Dongarra）等在算法容错技术基础上所作的进一步推进，以处理高性能计算应用中的节点失效问题。该技术将计算节点上的数据编码（通常是做冗余和）存储在额外的冗余节点上，并设计并行算法，使得冗余节点和计算节点上的数据在计算过程中同步更新，从而实现对失效节点上数据的恢复。

考虑单节点失效的情况。由于在节点失效发生之前，我们并不知道哪个节点将会失效，因此一套容错方案必须能提供恢复任意一个节点上数据的机制。假定有 $\boldsymbol{n}$ 个计算节点。每个计算节点上的数据为 $\boldsymbol{D_i}$，冗余节点上的数据为 $\boldsymbol{E}$。在计算过程中，各节点上的数据满足如下关系

$$\boldsymbol{D}_1 + \boldsymbol{D}_2 + \cdots + \boldsymbol{D}_n = \boldsymbol{E} \tag{1}$$

如果节点 $\boldsymbol{i}$ 失效，则 $\boldsymbol{i}$ 上的数据 $\boldsymbol{D_i}$ 可由下式恢复

$$\boldsymbol{D}_i = \boldsymbol{E} - (\boldsymbol{D}_1 + \cdots + \boldsymbol{D}_{i-1} + \boldsymbol{D}_{i+1} + \cdots + \boldsymbol{D}_n) \tag{2}$$

然而，在实际应用中，这种冗余和关系的保持并不是固有的，而是需要特定的设计。如何设计算法使得冗余和关系在计算过程中可以保持，也是研究算法容错需要解决的问题。

2.2 MPICH2 的容错特性

[3] High Performance Linpack，高性能 Linpack 标准测试程序，详见本文§3.3.1

为了实现算法级容错技术，MPI 实现必须提供一套机制，使用户能有效地参与容错。本文的实现使用 MPICH2-trunk-r7834 软件包，它的主要的容错特性有：

1. 节点失效不会导致整个程序终止。这一点通过在 `mpirun` 命令运行程序时加入 `-disable-auto-cleanup` 选项可以保证。
2. 算法级容错方案依赖底层对失效的检测。因此，MPI 实现必须能够提供一套机制，使得进程能够查询哪些节点失效。MPICH2 实现对 `MPI_COMM_WORLD` 通信子新增了 `MPICH_ATTR_FAILED_PROCESS` 属性。进程可以通过查询该属性的值，来获知节点失效信息。
3. 包含失效节点的通信操作不会挂起。大量的测试表明，MPICH2 已能做到这一点。
4. 受失效节点影响的通信操作会返回错误码，主要用于决定消息是否需要重发和重接收。需要说明的是，为了降低无失效情况下集合操作的开销，MPICH2 降低了对集合通信的限制。比如包含失效进程的 `barrier`（栅栏）操作并不能保证所有进程都能返回错误码；包含失效进程的 `bcast`（广播）操作也不能保证所有进程都正确地收到了消息。

需要说明的是，MPICH2-trunk-r7834 不支持在节点失效的情况下动态调整通信子，但是最新版本的 MPICH2-1.4 可以做到这一点。在本文的实现中，通信子的调整需要在应用层上实现。我们将在第四章中涉及这个问题。

3 方案设计

本章提出了一种新型高效的算法级容错技术，探讨了一次失效和多次失效的处理方案，最后针对 HPL 做了相应的容错设计。

3.1 热替换策略

为简单起见，这里先考虑一次失效的情况。假定在计算过程中，计算节点 $\boldsymbol{i}$ 上的数据 $\boldsymbol{D_i}$ 和冗余节点上的数据 $\boldsymbol{E}$ 满足如下关系

$$\boldsymbol{D}_1 + \boldsymbol{D}_2 + \cdots + \boldsymbol{D}_n = \boldsymbol{E} \tag{3}$$

一旦节点 $\boldsymbol{i}$ 失效，我们不是让所有“活着”的进程停下来等待失效进程上数据的恢复，而是用冗余节点替换失效节点，使计算继续进行下去。

从全局的角度看，发生失效前，$\boldsymbol{n}$ 个计算节点上的数据为

$$\boldsymbol{D} = (\boldsymbol{D}_1, \cdots, \boldsymbol{D}_{i-1}, \boldsymbol{D}_i, \boldsymbol{D}_{i+1}, \cdots \boldsymbol{D}_n) \tag{4}$$

替换后变为

$$\boldsymbol{D}' = (\boldsymbol{D}_1, \cdots, \boldsymbol{D}_{i-1}, \boldsymbol{E}, \boldsymbol{D}_{i+1}, \cdots \boldsymbol{D}_n) \tag{5}$$

令 $\boldsymbol{D}' = \boldsymbol{D} \times \boldsymbol{T}$，则 $\boldsymbol{T}$ 可以表示成一个 $\boldsymbol{n} \times \boldsymbol{n}$ 的矩阵，如下所示：

$$\boldsymbol{T}=\begin{pmatrix} 1 & & & 1 & & & \\ & \ddots & & \vdots & & & \\ & & 1 & 1 & & & \\ & & & 1 & & & \\ & & & 1 & 1 & & \\ & & & \vdots & & \ddots & \\ & & & 1 & & & 1 \end{pmatrix} \tag{6}$$

其中，对角线和第 $\boldsymbol{i}$ 列省略的元素为 1，其余为 0。

从（6）可以看出，$\boldsymbol{T}$ 是一个非奇异矩阵。如果对数据 $\boldsymbol{D}$ 的所有操作均是线性变换（如

矩阵的 LU 分解等），则 $\boldsymbol{D'} = \boldsymbol{D} \times \boldsymbol{T}$ 的关系可以一直保持。在运算的最后，基于原始数据 $\boldsymbol{D}$ 的正确结果可以由一个基于 $\boldsymbol{D'}$ 的中间解求出。这个过程实际上是一个基于 $\boldsymbol{T}$ 的线性变换。

然而，$\boldsymbol{D'} = \boldsymbol{D} \times \boldsymbol{T}$ 的这种编码关系并不能在所有高性能计算应用中都保持，但在一类包含线性变换的矩阵运算中，如矩阵的加法、乘法、LU 分解、内积和转置等，这种关系可以保持。

3.2 多次失效处理

对于多次失效的情况，一种方案是使用多级冗余。但对于下一代高性能系统来说，该方案并不能有效地解决这个问题，因为冗余总有耗尽的时候。

另一个新的方案是通过后台加速重建冗余和，即对替换后的数据进行重新编码，保存在新的冗余节点上。重建冗余和涉及大量的通信操作，并且需要计算节点参与。为此我们提出了 “后台加速重建”，就是使计算节点尽早地从重建冗余的工作中“解放”出来，通过增加节点或网络在后台加速重建冗余，并使冗余节点能尽快“追上”计算节点。而正确结果可以通过增加若干次热替换后计算得到中间结果，再多次迭代变换而得到。

3.3 针对 HPL 的容错设计

3.3.1 HPL 概述

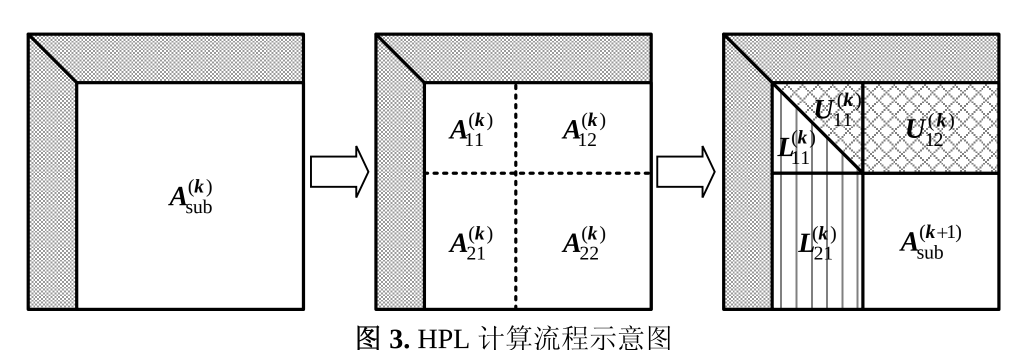


图 3. HPL 计算流程示意图

Each process generates its local random matrix A
`for k = 0, 1, …`
　calculate $\boldsymbol{A}_{11}^{(k)} = \boldsymbol{L}_{11}^{(k)} \times \boldsymbol{U}_{11}^{(k)}$ and $\boldsymbol{A}_{21}^{(k)} = \boldsymbol{L}_{21}^{(k)} \times \boldsymbol{U}_{11}^{(k)}$;　　消元
　broadcast $\boldsymbol{L}_{11}^{(k)}$, $\boldsymbol{L}_{21}^{(k)}$ and pivoting information right;　　行广播
　perform row swaps and calculate $\boldsymbol{U}_{12}^{(k)} = \boldsymbol{L}_{11}^{(k)-1} \times \boldsymbol{A}_{12}^{(k)}$;　　更新
　update the trailing sub-matrix $\boldsymbol{A}_{sub}^{(k+1)} = \boldsymbol{A}_{22}^{(k)} - \boldsymbol{L}_{21}^{(k)} \times \boldsymbol{U}_{12}^{(k)}$;　　更新
solve $\boldsymbol{Ux} = \boldsymbol{L}^{-1}\boldsymbol{b}$ to obtain $\boldsymbol{x}$;　　回代

图 4.　HPL 程序的伪代码

HPL 是 TOP500 超级计算机排名所用的基准测试程序。它的主要算法是用列主元高斯消元法（Gaussian Elimination with Partial Pivoting）求解线性方程组 $\boldsymbol{Ax}=\boldsymbol{b}$。它首先计算 $\boldsymbol{n}\times(\boldsymbol{n}+1)$ 阶系数矩阵$[\boldsymbol{A}|\boldsymbol{b}]$的 LU 分解，得到$[\boldsymbol{A}|\boldsymbol{b}] = [[\boldsymbol{L},\boldsymbol{U}]\ \boldsymbol{L}^{-1}\boldsymbol{b}]$。然后解方程组 $\boldsymbol{Ux} = \boldsymbol{L}^{-1}\boldsymbol{b}$ 求解 $\boldsymbol{x}$，其中 $\boldsymbol{U}$ 是上三角矩阵，$\boldsymbol{L}$ 是下三角矩阵，见图 3。图 4 为使用 LU 分解算法时 HPL 程序的伪代码。更多信息，请参见[6]。

3.3.2 容错设计

下面先考虑一次节点失效的情况。如§3.3.1 所述，HPL 是用列主元高斯消元法求解线性方程组

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{7}$$

假定进程 $\boldsymbol{P_i}$ 在程序运行中失效，则使用冗余数据替换失效数据后，线性方程组变成

$$\boldsymbol{A}'\boldsymbol{y} = \boldsymbol{b} \tag{8}$$

在程序执行的最后，得到中间解 $\boldsymbol{y}$。由于列主元高斯消去法只涉及线性变换，因此

$$\boldsymbol{A}' = \boldsymbol{A} \times \boldsymbol{T} \tag{9}$$

的关系在程序执行过程中可以一直保持。

联立（7）、（8）和（9），解 $\boldsymbol{x}$ 可以由下式求出

$$\boldsymbol{x} = \boldsymbol{T} \times \boldsymbol{y} \tag{10}$$

假定变换矩阵 $\boldsymbol{T}$ 具有如（6）所示的形式，则将（6）代入（10）可得

$$\begin{cases} x_j = y_i + y_j, & 1 \le j \ne i \le n \\ x_i = y_i \end{cases} \tag{11}$$

由上式可以看出，由 $\boldsymbol{y}$ 计算 $\boldsymbol{x}$ 的代价为 $O(n)$，相比算法失效恢复技术处理单节点失效的开销 $O(n^2)$ 大大降低。

对于多次失效的情况，可以通过迭代由中间解 $\boldsymbol{y}$ 求得 $\boldsymbol{x}$。考虑 $\boldsymbol{n}$ 次失效的情况，第 $\boldsymbol{i}$ 次失效的变换矩阵为 $\boldsymbol{T_i}$，则

$$\boldsymbol{x} = \boldsymbol{T}_1 \times \boldsymbol{T}_2 \times \cdots \boldsymbol{T}_n \times \boldsymbol{y} \tag{12}$$

在具体实现中，由于 HPL 使用二维的块循环（block-cyclic）数据分布，矩阵的编码和变换矩阵会更复杂，我们将在下一章涉及这些内容。

3.4 优点

和算法失效恢复技术相比，我们的方案有如下优点：首先，热替换策略使得故障瞬时切换，不需要停等恢复失效节点；其次，由于变换矩阵 $\boldsymbol{T}$ 是一个非常稀疏的矩阵，由中间结果计算最终解的开销很小，远低于算法失效恢复技术恢复失效节点上数据的开销。正是基于以上两点，算法失效热替换方案在理论上性能要优于算法失效恢复技术。为了验证方案的正确性，下一章将阐述把算法失效热替换应用到 HPL 中的实现细节。

4 方案实现

目前，我们已实现使用算法失效热替换方案处理单节点失效的容错的 HPL。以下实现均是针对单节点失效的情况。如§3.3.1 所述，HPL 的主体部分是一系列消元、行广播和更新阶段的叠加。它们占据了 HPL 总执行时间的绝大部分。我们的工作也是针对这段时间的容错，因此如果节点失效发生在这段时间之外，HPL 的所有进程都会终止。

4.1 矩阵编码

矩阵的编码使用了和引文[4]类似的方案。在 HPL 中，随机矩阵 $\boldsymbol{A}$ 按照块循环模式分布在一个二维 $\boldsymbol{P}\times\boldsymbol{Q}$ 的进程网格上。为了处理单节点失效，我们新增了 $\boldsymbol{P}$ 个进程。将它们和原

来的进程一起组成一个新的(***P***+1)× ***Q*** 的网格。冗余的 ***P*** 个进程分布在新网格的第 ***Q***+1 列上，存储前 ***Q*** 列进程数据的编码。编码信息可以由前 ***Q*** 列进程的局部数据相加得到。图 5 为二维进程网格及编码数据的分布。如图 5(a)所示 ***P***=2、***Q***=2 的二维进程网格加入冗余进程后，组成如图 5(b) 所示的新的进程网格。

在这种编码模式下，对矩阵 ***U*** 的"行和"关系可以在计算过程中保持，即在每次更新阶段完成后保持 [16]。

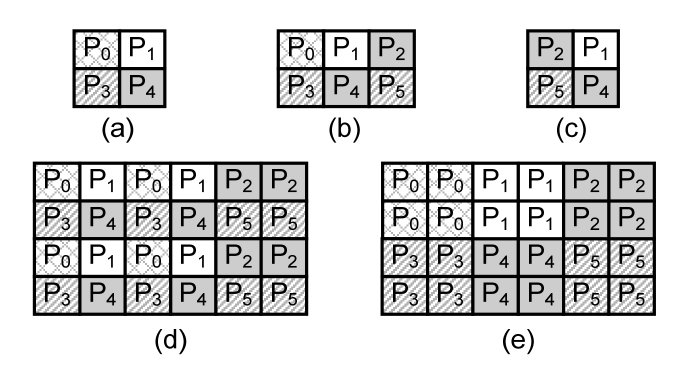


图 5. 二维进程网格及编码矩阵的分布。
图中：（a）原始的进程网格；（b）包含冗余进程的网格；（c）P3 失效被替换后重组的进程网格（d）编码矩阵的全局视图；（e）编码矩阵的视图

4.2 失效检测和热替换

首先，我们给出优化后双精度通用矩阵乘法（DGEMM，Double-precision General Matrix Multiply）在异构系统上的整体性能。性能对比的基准程序对失效的发生是无法预测的，因此，在失效发生前，我们不知道何时会发生失效、哪个节点将失效。这些信息可以通过定期（在每个阶段之后）查询 `MPI_COMM_WORLD` 的 `MPICH_ATTR_FAILED_PROCESSES` 得到。由于不同的进程完成同一阶段的时间不一致，因此在查询之前需要执行一个栅栏（barrier）操作，以保证查询结果的一致性。

如果在某一个阶段后发现有一个进程失效了，则所有"活着"的进程进入一个统一的入口使用算法失效热替换方案处理失效，即使用冗余进程列替换失效进程列，使计算继续进行下去。以图 5(b)的进程网格为例，假设进程 P_3 失效，则热替换后进程网格如图 5(c)所示。

这里的热替换并不是物理上的替换，而是通过交换进程网格相关的数据结构中某些成员变量实现的。它不包含任何数据通信，因此可以迅速地完成。

需要说明的是，热替换只能发生在 ***U*** 的"行和"关系保持的前提下，即在更新阶段完成后。如果在某个消元阶段后发现进程失效，则剩余"活着"的进程需要完成行广播和更新之后，才能进行热替换。

另一个问题是如何更新热替换后的通信子状态。通信子是用来表示一组"活着"的进程。由于 MPICH2 暂时不能动态地调整通信子，如加入或移除某些进程等。为了解决这个问题，我们在应用层为每一个进程指定一个虚拟的序。进程之间使用虚拟序进行通信。为此，我们写了两个宏，根据处理机网格的状态信息实现虚拟序和进程在 `MPI_COMM_WORLD` 中的序之间的映射。

4.3 后台恢复

可以发现，热替换之后，在最后一块尾随矩阵更新完成后得到的 ***U*** 不再是上三角矩阵。这为我们计算 $Uy=L^{-1}b$ 求解 ***y*** 带来困难。一个解决办法是，对于已经分解过的数据，我们不用冗余数据替换它，而是将它恢复在相应的冗余进程上。这样做的另一个好处是，这些数据不是立即要用的，而是在最后的回代环节才会用到，因此这些数据的恢复可以在后台进行。我们使用非阻塞的点对点通信，将恢复所需的通信部分和随后更新阶段的计算部分重叠起来。对于恢复数据量比较大的情况，我们可能需要将后台恢复的数据分成若干部分。需要注意的是，随着计算的进行，需要更新的数据量在不断减少，从而每次划分出的通信数据量也

应越来越少。

4.4 消息的故障处理

上述方法足以处理消元和更新阶段发生节点失效的情况。对于行广播阶段，我们需要特殊的处理。HPL 提供的 6 种行广播方式都是基于消息转发（message forwarding）的。如果在广播中有一个进程失效，则会导致那些不和它直接通信的进程挂起。假如进程 $\boldsymbol{P}_1$ 转发由进程 $\boldsymbol{P}_0$ 发送来的消息给进程 $\boldsymbol{P}_2$，$\boldsymbol{P}_0$ 失效了。$\boldsymbol{P}_1$ 可以通过 `MPI_Recv` 的返回错误码获知 $\boldsymbol{P}_0$ 已失效，但 $\boldsymbol{P}_2$ 却无法获知该信息，因为它的上一级节点 $\boldsymbol{P}_1$ 仍“活着”。因此，对于广播过程中的进程失效，我们需要一套健壮的消息广播机制。它应满足如下两个条件：

1. 节点失效不会引起参与广播的任何节点挂起；
2. 要么所有节点都成功接收到消息，要么都返回出错信号 error。

本节我们针对 HPL 一种常用的广播方式 increasing-2-ring-modified(2rinM)，定制“健壮”的广播方式 。

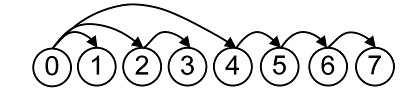


图 6. 2rinM 方式下消息传递路径

2rinM 方式下消息传递的路径如图 6 所示。即首先由进程 0 传给进程 1，剩下的 $\boldsymbol{Q}$-1 进程被分成两组：进程 2~($\boldsymbol{Q}$+1)/2-1 为一组，进程($\boldsymbol{Q}$+1)/2~($\boldsymbol{Q}$-1)为另一组。接着以进程 2 和进程($\boldsymbol{Q}$+1)/2 为源节点顺次传递消息。

按照消息传输的时间先后关系，将图 6 中的消息传递表示成树状如图 7 所示。我们为每个非根节点指定一个父亲节点。父节点的指定需要满足两个条件：(1). 父节点的消息传递先于子节点； (2). 父子节点的消息传递没有直接的依赖关系。父子节点每次退出 `MPI_Recv` 后握手，以决定是否重发和重接收消息。即每一个非根节点将 `MPI_Recv` 的返回码发送给它的父节点。如果 `MPI_Recv` 返回出错信息，它还需要等待父节点重发的消息。对于一个父节点来说，它接收子节点的返回码，并判定是否需要向子节点重发消息。以图 7 为例，我们按图中所示的方式指定父节点，图中的虚线由子节点指向相应的父节点。按照这种方法，我们可以保证对单节点失效而言，任何一个非根节点的失效都不影响消息成功的传递。

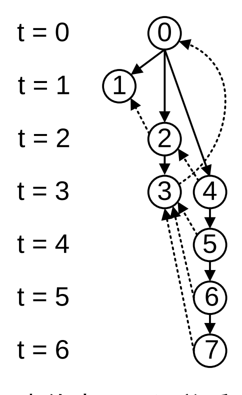


图 7. 2rinM 方式下的消息传递树状图

但如果是根节点 0 在第一条消息传输成功之前就失效了呢？如此，则所有不直接接收 0 的消息的节点都会挂起。为了解决这个问题，我们将第一条消息的传输从行广播中分离开来。仅当第一条消息传输成功后，才使用上述机制。

5 方案验证

本章我们通过实验对算法失效热替换方案的性能和舍入误差进行了评估。针对如下问题，进行了三组实验。

1. 算法失效热替换方案性能如何？
2. 算法失效热替换方案对计算精度的影响如何？
3. 失效时刻对性能有什么影响？

前两组实验在曙光 5000A 平台上开展，共使用 16 个节点。每个节点包含 4 个 quad-core 2.2GHz AMD Opteron 处理器，共享 64GB 内存。节点之间由千兆以太网相连。第三组实验在 8 个刀片组成的“超龙”平台上开展。每个刀片包含 10 个 Intel Xeon X5650 处理器，共计 960 个核。同一个刀片内的节点由 Infiniband 相连，而两个刀片之间由一根 Infiniband 相连。这两个平台上的节点都运行 Linux 操作系统。实验中使用的 MPICH2 软件包是 MPICH2-trunk-r7834。所有的计时是通过调用 `MPI_Wtime` 函数统计的。

5.1 算法级容错方案性能的比较

表1. D5000A 平台上的实验配置

矩阵阶数	节点数	每个节点上的进程数	P×Q
10,000	1	6	6×1
20,000	1	16	4×4
30,000	2	16	4×8
40,000	4	16	16×2
50,000	12	16	16×12
60,000	16	16	16×16

表2. HPL 的执行时间

矩阵阶数	无失效	热替换	恢复
10,000	577.74	591.61	602.73
20,000	1608.15	1677.14	1723.37
30,000	2601.03	2791.12	2821.43
40,000	3150.11	3358.38	3497.08
50,000	3433.09	3479.26	3577.77
60,000	4708.54	4722.44	4858.23

第一组实验是用来比较两种不同的算法级容错方案的性能。我们将算法失效恢复技术和算法失效热替换方案分别集成到 HPL 中去处理单节点失效。节点失效是通过强行终止一个进程来模拟的。

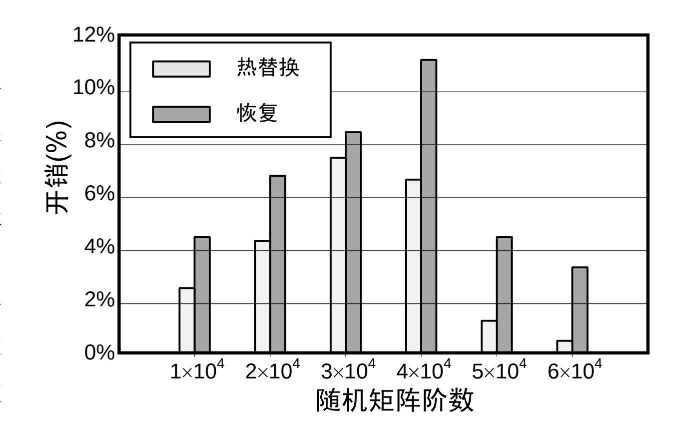


图 8. 不同算法级容错方案的开销

在本组实验中，随机矩阵 ***A*** 的阶数及计算节点的使用情况见表 1。矩阵的阶数为 10^4 级。使用算法失效恢复技术和算法失效热替换方案处理单节点失效和无失效情况下 HPL 的执行时间，如表 2 所示。两种方案处理单节点失效相对于无失效情况下的开销如图 8 所示。从表 2 可看出，处理单节点失效，算法失效热替换方案与算法失效恢复技术方案相比，HPL 的总执行时间减少了 10~200 秒。这部分时间相当无失效情况下 HPL 总执行时间的 1%~5%。两种方案下的开销除了会随单个进程分配到的数据量的多少而变化，还会受进程和处理器核之间的映射关系影响。

5.2 算法级容错方案舍入误差的比较

算法级容错方式通过对浮点数进行算术编码构造冗余数据，引入了一定的舍入误差。此外，算法失效热替换使用冗余数据替换失效数据，涉及到矩阵变换，进一步加剧了舍入误差。第二组实验正是为了衡量该方案带来的舍入误差问题。舍入误差通过检查 HPL 误差检验结果得到，即（其中 N 是矩阵维数，ε是机器精度）：

$$\frac{\|Ax-b\|_{\infty}}{\varepsilon(\|A\|_{\infty}\times\|x\|_{\infty}+\|b\|_{\infty})\times N} \qquad (13)$$

本组实验使用的数据规模和节点信息同第一组实验相同，见表 1。图 9 为使用算法失效热替换和算法失效恢复技术处理单节点失效以及无失效情况下的 HPL 误差检验结果。可以看出，算法失效恢复技术方案引入的舍入误差较小，而算法失效热替换带来的舍入误差大致是无失效情况下的 2 倍。

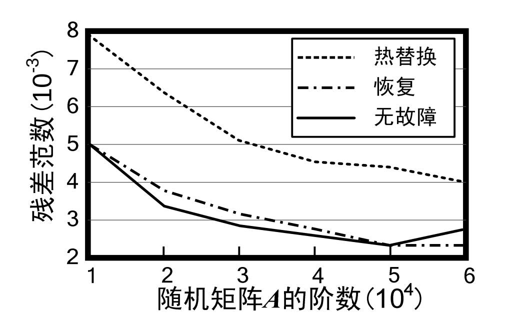


图 9. 不同容错方式引起的误差比较

5.3 不同失效时刻对热替换方案性能的影响

第三组实验是为了评估失效时刻（timing）对算法失效热替换方案性能的影响。本组实验是在“超龙 1 号”平台上进行的。我们使用了其中的 75 个节点，共计 900 个核。我们在每个核上启动一个进程，并把这 900 个进程组成一个 30×30 的网格。本组实验中矩阵阶数为 20,000，失效时刻的控制由 python 脚本中的 `sleep` 函数实现。

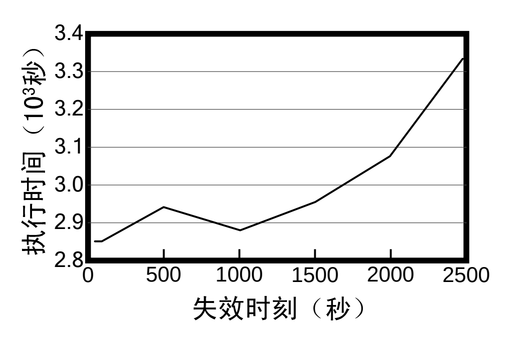


图 10. 失效时刻对性能的影响

实验结果如图 10 所示。可以看出，在计算开始 1,000 s 后失效，HPL 的总执行时间急剧上升。这是由于随着计算的进行，需要后台恢复的数据量也在增加。当数据量增加到一定程度，以致不能完全被随后的更新阶段重叠起来时，就会带来额外的开销。不过，从理论上讲，算法失效热替换方案下带来的最大开销也不会超过算法失效恢复技术方案。

6 工作展望

下一步工作有三个可能的方向：

第一个方向是设计后台加速重建冗余和的方案，以处理多次节点失效的情况。该方案最大的难点在于开销问题。重建冗余和涉及大量的通信，且需要计算节点参与。因此我们需要设计高效的方案使得计算节点能尽快地从重建冗余和的工作中“解放”出来，并使冗余节点能尽快地“追上”计算节点。另一个问题是热替换策略带来的舍入误差问题。图 11 显示了舍入误差随失效次数变化的情况。实验表明，当失效次数超过 10 次时，计算结果往往通不过 HPL 的误差检验。选择更具数值稳定性的编码方案或者通过经验值校正最终结果可能成为我们以后的工作之一。此外，多次失效的实现，除了在应用层做适当的改动外，还依赖 MPI 实现新的支持，如动态地调整通信子等。

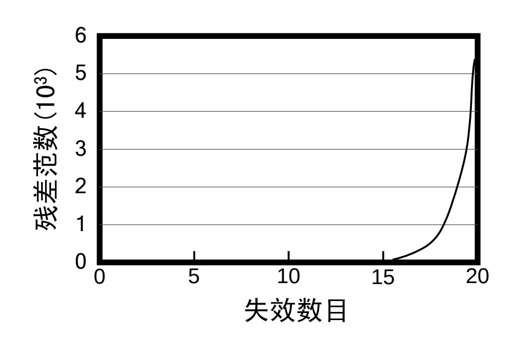


图 11. 舍入误差与失效次数的关系

将方案应用到更大规模的超级计算机上，实现更完备的失效处理是我们第二个研究方向。此外，将热替换策略扩展到更多的高性能计算应用中也是我们今后要研究的问题。

参考文献：

[1] The International Exascale Software Project. http://www.exascale.org.

[2] Top 500 supercomputing sites. http://www.top500.org. Last accessed February 2011.

[3] Z. Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In Proceedings of the *20st IEEE International Parallel and Distributed Processing Symposium*, page 76, 2006.

[4] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12), December 2008.

[5] Z. Chen, G. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Building fault survivable mpi programs with ft-mpi using diskless checkpointing. *In Proceedings for ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–223, 2005.

[6] HPL benchmark sites. http://www.netlib.org/benchmark/hpl.

[7] K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

[8] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78, 2007.

[9] MPI-Forum fault-tolerance working group. https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_users_guide. .

[10] Franck Cappello. Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *International Journal of High Performance Computing Applications*, 23:212-226, August 2009

[11] G. Gibson, B. Schroeder, and J. Digney. Failure tolerance in petascale computers. *CTWatchQuarterly*, 3(4), November 2007.

[12] G. Gibson, Reflections on Failure in Post-Terascale Parallel Computing, Keynote at Int. Conf. on Parallel Processing, Xi'An China, 2007.

[13] The computer failure data repository sites. http://cfdr.usenix.org.

[14] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, 1998.

[15] D. Hakkarinen and Z. Chen. Algorithmic Cholesky factorization fault recovery. In Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium, Atlanta, GA, USA, April 2010.

[16] T. Davies, C. Karlsson, H. Liu, and Z. Chen. Algorithm-based recovery for HPL. In Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 303–304, 2011.

[17] Z. Chen and J. Dongarra. Highly scalable self-healing algorithms for high performance scientific computing. *IEEE Transactions on Computers*, July 2009.

[18] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. Proceedings of the *20th ACM International Symposium on High-Performance Parallel and Distributed Computing*, June 2011.

作者简介：

王　睿： 体系结构国家重点实验室、硕士研究生
姚二林： 体系结构国家重点实验室、助理研究员
陈明宇： 体系结构国家重点实验室、研究员 cmy@ict.ac.cn
谭光明： 体系结构国家重点实验室、副研究员